# PART I

# Background

# CHAPTER 2

# An Introduction to Neural Networks

**Abstract** | Deep Neural Networks are at the forefront of many state-of-the art approaches to Natural Language Processing (NLP). The field of NLP is currently awash with papers building on this method, to the extent that it has quite aptly been described as a tsunami (Manning, 2015). While a large part of the field is familiar with this family of learning architectures, it is the intention of this thesis to be available for a larger audience. Hence, although the rest of this thesis assumes familiarity with neural networks, this chapter is meant to be a foundational introduction for those with limited experience in this area. The reader is assumed to have some familiarity with machine learning and NLP, but not much beyond that.

We begin by exploring the basics of neural networks, and look at the three most commonly used general architectures for neural networks in NLP: Feed-forward Neural Networks, Recurrent Neural Networks, and Convolutional Neural Networks. Some common NLP scenarios are then outlined together with suggestions for suitable architectures.

## 2.1 Introduction

The term *deep learning* is used to refer to a family of learning models, which represent some of the most powerful learning models available today. The power of this type of model lies in part in its intrinsic hierarchical processing of input features, which allows for learning representations at multiple levels of abstraction (LeCun et al., 2015). This type of model is commonly referred to by several umbrella terms, such as *deep learning*, and *(deep) neural networks*.[1] In this chapter, I aim to introduce the basic concepts of NNs, at a level sufficient to understand the work in this thesis. The following sections are meant to cover the most basic workings in an intuitive, and theoretically supported, manner. In addition to this background, I give an overview of the scenarios in which different recurrent NN architectures might be suitable in NLP (Section 2.4.2).

### History

Neural networks have a long history, and have been popular in three main waves.[2] In the first wave, roughly between the $40s - 60s$, they appeared under the moniker of *cybernetics* (e.g., Wiener, 1948), inspired by the Hebbian learning rule (Hebb, 1949). In this wave, the *perceptron* was first outlined (Rosenblatt, 1957), which is still relatively popular today (see Section 2.3). Next, in the $80s$ and $90s$, *connectionism* was on the rise. In this wave, the algorithm for backwards propagation of errors (Rumelhart et al., 1985) was described, which is at the core of how neural networks are trained (see Section 2.3.3).

[1] While some make a distinction between *deep* and non-deep neural networks (NNs) depending on the amount of layers used, there is no real consensus on where the line between these models should be. For the sake of consistency, I attempt to refer to this family of models as NNs, or some specification thereof, as consistently as possible.

[2] Only a very brief overview of the history is given here. For more details, the reader is referred to, e.g., Wang et al., 2017, or Goodfellow et al., 2016.

After an AI winter lasting roughly from $97$ to $06$, we finally arrive at the current wave (or tsunami), in which the term *deep learning* is favoured. This wave was initiated by works on deep belief networks (Hinton et al., 2006), and has been the subject of much attention after successes in, e.g., reducing error rates in some tasks by more than 50% (LeCun et al., 2015). The recent advances made in the current wave further include breakthroughs in both recognition (He et al., 2016) and generation (Goodfellow et al., 2014) of images, in NLP tasks such as machine translation (Bahdanau et al., 2015; Wu et al., 2016) and parsing (Chen and Manning, 2014), as well as in the strategic board game Go (Silver et al., 2016), and the first-person shooter Doom (Lample and Chaplot, 2017).

## 2.2 Representation of NNs, terminology, and notation

Before embarking upon this journey and exploring the wondrous world of NNs, it is necessary to equip ourselves with some common ground in terms of terminology, notation, and how NNs are generally represented in this thesis. Figure 2.1 contains a NN, which we will go through in detail.

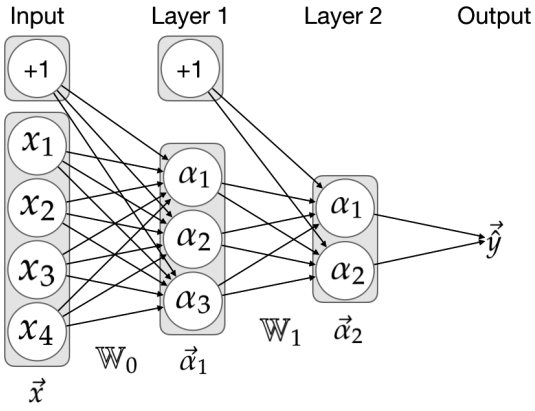


Figure 2.1: A basic Neural Network.

First, note that the network is divided into three vertical slices. Each such slice represents a *layer*, marked by a light grey field. Each layer contains one or more white circles, each representing a *unit*, or *neuron*.[3] In this network, each unit has a connection to every unit in the following layer. These connections are represented by arrows, which denote some weighting of the output of the unit at the start of the arrow, for the input of the unit at the end of the arrow. Each layer can be described mathematically as a vector of *activations*. In the case of the first layer (the input layer), these activations are equal to the input (i.e. $\vec{\alpha}_0 = \vec{x}$). The final layer encodes the output of the network, which is denoted by $\hat{y}$. Each layer up until the final layer also contains a special unit, marked by $+1$, which is called the *bias* unit.[4] The collection of all arrows between two layers, can be described mathematically as a matrix of weights (e.g., $\mathbb{W}_0$).[5] The application of the weight matrix to the input of the network can be described in linear algebraic notation as

$$\vec{z}_1 = \mathbb{W}_0\vec{x} = \mathbb{W}_0\vec{\alpha}_0, \tag{2.1}$$

where $\vec{z}_1$ is the vector (i.e. a series of numbers) resulting from this linear transformation. The $\vec{z}$-vectors can be referred to as pre-activation vectors. Each hidden layer thus first encodes the sum of the multiplications of each of the activations in the previous layer by some weight. For instance, if we set $\mathbb{W}_0$ to be matrix of ones, then the pre-activation value of the first unit in the first hidden layer $z_0^1 = \Sigma_{i=0}^4 \vec{x}_i$. The final piece of the puzzle is to calculate the output of each unit in the layer, by applying an activation function to the pre-activation

[3] In this thesis, the term *unit* is preferred. While this is conventional in much of NLP, it is also debatable whether borrowing terminology for neural networks from neuroscience is motivated at all (this is discussed further in Section 2.7).

[4] Although this can be discussed at length, suffice it to say that including bias units facilitates learning.

[5] We will cover ways in which to learn these weights later in this chapter (Section 2.3.3).

vector,

$$\vec{\alpha}_1 = \sigma(\vec{z}_1), \tag{2.2}$$

where $\sigma$ is some non-linear activation function.[6] Essentially, this is all that a basic FFNN is – a series of matrix-vector multiplications, with non-linearities applied to it. Now, how can this be used to solve problems, and how does the network learn to do this? The answers to these questions will be made clear in the course of the following few pages.

### Notation

Before we continue, a brief note on the notation used in this thesis. Scalars are represented with lower case letters ($x, y$), vectors are represented with lower case letters with arrows ($\vec{x}, \vec{y}$), and matrices are represented with blackboard upper case letters ($\mathbb{X}, \mathbb{Y}$). Subscripts are used to denote the layer number, and where necessary, a superscript is used to denote indexation, to indicate the unit number in the case of deep networks. For instance, $\alpha_i^j$ indicates the activation of unit $j$ in layer $i$, and $\mathbb{W}_i$ indicates the weight matrix for layer $i$. Activation functions (Section 2.3.2) are denoted with $\sigma$, occasionally subscripted with the actual function used ($\sigma_{ReLU}$).

## 2.3 Feed-forward Neural Networks

A Feed-forward Neural Network (FFNN), also known as a multilayer perceptron, is perhaps the most basic variant of neural networks, and is the kind depicted in the previous figure. As mentioned, a neural network can be seen as a collection of non-linear functions applied to a collection of matrices and vectors, thus mapping from one

[6] Traditionally the activation function ($\sigma$) used is some *sigmoidal* function, such as the logistic function. However, many functions are suitable, given that they satisfy certain properties (see Section 2.3.2).

domain (e.g., words) to another (e.g., PoS tags). Let us consider a concrete example, in which $\mathbb{X}$ contains information about the current weather, and $\mathbb{Y} = \{0, 1\}$ denotes human-annotated labels denoting whether or not the weather is considered good. In this case, $\mathbb{X}$ contains several variables, each representing a certain type of weather (e.g. $x_1$ = calm weather, and $x_2$ = sunny weather).[7] Table 2.1 represents the weather judgements of this example, where $y = 1$ indicates good weather, and $y = 0$ indicates not-so-good weather.

Table 2.1: Weather appraisal (mimicking the logical AND function).

| **Calm ($x_1$)** | **Sunny ($x_2$)** | **Label ($y$)** |
|---:|---:|---:|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

The table shows the judgements of someone who considers weather to be good (i.e. $y = 1$) only when it is both calm *and* sunny (i.e. the logical AND function). Let us now consider our second neural network, which can solve the problem of determining whether the weather is good, based on this person's judgements, in Figure 2.2.

Table 2.2: Weather appraisal by the neural network in Figure 2.2.

| **Calm ($x_1$)** | **Sunny ($x_2$)** | $z_1$ | $\sigma(z_1) = \sigma(\alpha_1) = \hat{y}$ | **Label ($y$)** |
|---:|---:|---:|---:|---:|
| 0 | 0 | $-15$ | $\approx 0$ | **0** |
| 0 | 1 | $-5$ | $\approx 0$ | **0** |
| 1 | 0 | $-5$ | $\approx 0$ | **0** |
| 1 | 1 | 5 | $\approx 1$ | **1** |

Applying the calculations detailed in the previous section to this net-

[7] Note that we begin numbering of features with 1, as the index 0 is reserved for the bias terms.

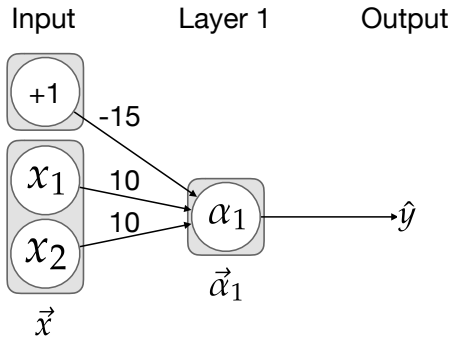


Figure 2.2: A Neural network coding the AND logical function.

work yields the results shown in Table 2.2. If only one of $x_1, x_2$ is active, the activation $\alpha_1$ is approximately $0$, while the activation is $1$ if both $x_1$ and $x_2$ are active. We get these values, by applying the perhaps most commonly used activation function to $z$. This is the logistic function, defined as

$$f(z) = \frac{1}{1 + e^{-z}}, \tag{2.3}$$

where $e$ is Euler's number. Plotting this function, yields the graph in Figure 2.3. Hence, the value of $f(x)$ approaches 0 when $x < 0$, and approaches 1 when $x > 0$.

The simple neural network considered here, is what is also referred to as a *perceptron*, although, technically, a perceptron uses the *step* function as its activation function – in other words, if $x < \lambda$ where $\lambda$ is some threshold, then $f(x) = 0$, and if $x > \lambda$, then $f(x) = 1$ (Rosenblatt, 1957). This is a very simple and useful architecture, but there are many problems which can not be easily solved by a perceptron, such as those in which the decision boundary to be learned is non-linear. Take, for instance, the problem given in Table 2.3.

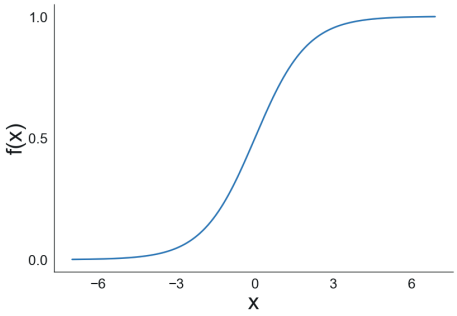


Figure 2.3: Plot of the logistic function.

Table 2.3: Weather appraisal (mimicking the logical XOR function).

| **Snowy ($x_1$)** | **Sunny ($x_2$)** | **Label ($y$)** |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

This table shows the labels provided by some annotator who considers weather to be good (i.e. $y = 1$) if it is *either* snowy *or* sunny – but not both (i.e. the logical XOR function). As mentioned, a single unit (i.e. a perceptron) is not able to learn this decision boundary (Minsky and Papert, 1988).[8] Let us now have a look at a neural network which encodes this function. This is depicted in Figure 2.4. For the sake of clarity, all weights between $x_1, x_2$ and $a_1, a_2$ are set to $5$, but only one of these weights is shown.

Applying the calculations detailed in the previous section to this

[8]This is frequently cited as a potential catalyst for the *the AI winter*, in which funding and interest in artificial intelligence was at a low. Nonetheless, it was known at the time that the XOR problem could be solved by a neural network with more hidden units (Rumelhart et al., 1985).

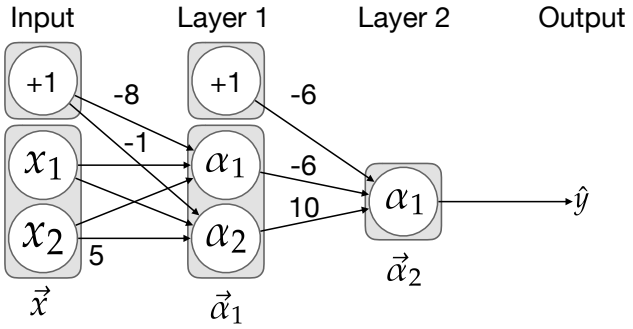


Figure 2.4: A Neural network coding the XOR logical function.

network yields the results shown in Table 2.4. If both or none of $x_1, x_2$ are active, then the network will output $0$, whereas if one and only one of $x_1, x_2$ are active, the network will output $1$. Exactly what we want!

Table 2.4: Weather appraisal by the neural network in Figure 2.4.

| **Calm ($x_1$)** | **Sunny ($x_2$)** | $z_1^1$ | $z_1^1$ | $z_2^1$ | $\sigma(z_2^1) = \alpha_1^2 = \hat{y}$ | $y$ |
|---|---|---|---|---|---|---|
| $0$ | $0$ | $-8$ | $-1$ | $-6$ | $\approx 0$ | **0** |
| $1$ | $0$ | $-3$ | $4$ | $4$ | $\approx 1$ | **1** |
| $0$ | $1$ | $-3$ | $4$ | $4$ | $\approx 1$ | **1** |
| $1$ | $1$ | $3$ | $9$ | $-2$ | $\approx 0$ | **0** |

The last two examples have shown how neural networks can encode certain simple functions. It turns out that neural networks can do much more than this, and are in fact *universal function approximators* (Cybenko, 1989; Hornik et al., 1989). What this means, is that no matter the function, there is guaranteed to be a neural network with a single layer and a finite number of hidden units, such that for each potential input $x$, (a close approximation of) the value $f(x)$ is output

from the network.

The class of networks discussed here is useful for many tasks in NLP, and can be used as a simple replacement for other classifiers. Furthermore, they can be expanded by adding more units to each layer, or by adding more layers. This allows such networks to learn to solve interesting NLP problems, like language modelling (Bengio et al., 2003; Vaswani et al., 2013), and sentiment classification (Iyyer et al., 2015).

Before going into other NN architectures, we will first consider some of the inner workings of NNs. This includes how we represent our input, how weights are obtained, and finally some limitations which motivate the use of more complex architectures than FFNNs.

### 2.3.1 Feature representations

In many NLP problems, we are interested in mapping from some textual language representation ($x$) to some label ($y$). This textual representation can take many forms, both depending on the problem at hand, and on the choices made when approaching the problem. As an example, say we are interested in doing sentiment analysis, i.e., given a text ($x$), predict whether the text is positive or negative in sentiment ($y$). The perhaps simplest way of representing the text is to count the occurrences of each word in the text. The intuition behind this is that if a text contains many negative words (*horrible, bad, appalling*), it is more likely to be negative in sentiment than if it contains many positive words (*wonderful, good, exquisite*). Since these *features* (i.e. counts of each word) need to be passed to an FFNN, they need to be represented as a single fixed-length vector $\vec{x}$. What one might then do, is to assign an index to each unique word, and assign the count of each word to that index in the vector. This can be referred to as a *bag-of-words* model.

Although this type of feature representation is sufficient for some problems, and is traditionally used extensively, more recent develop-

ments include using other types of representations based on distributional semantics. This is covered in more detail in the next chapter, in Section 3.2.5.

### 2.3.2 Activation Functions

As stated in Section 2.3, each hidden unit applies an activation function to the sum of its weighted inputs. While many functions might be used, an activation function should have certain properties. One such property is that the function needs to be non-linear. It is for this kind of function that it has been proven that a two-layer neural network is a universal function approximator (Cybenko, 1989). Additionally, the function should be monotonic, as the error surface associated with a single-layer model will then be convex (Wu, 2009).[9] There are several other important properties, which are not covered here. Some of the more commonly used activation functions are listed in Table 2.5.

Table 2.5: Commonly used activation functions in neural networks.

| **Name** | **Function** |
|---|---|
| Logistic (aka. sigmoid) | $f(x) = \frac{1}{1+e^{-x}}$ |
| Hyperbolic Tangent (tanh) | $f(x) = \frac{2}{1+e^{-2x}} - 1$ |
| Rectified Linear Unit (ReLU) | $f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$ |
| Leaky ReLU | $f(x) = \begin{cases} 0.01x & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$ |
| Softmax | $f(\vec{x})_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}$ for $i = 1, \ldots, K$ |

[9] A monotonic function is either non-increasing or non-decreasing in its entirety.

The traditionally popular logistic function was already described in Figure 2.3. We will now consider some other commonly used activation functions. Activation functions turn out to be one of the areas in which biological inspiration has been directly applicable to the development of neural networks. The Rectified Linear Unit (ReLU) is in fact remarkably similar to what happens in a biological neuron (Hahnloser et al., 2000; Hahnloser and Seung, 2001). That is to say, when the input is below a certain threshold, the neuron does not fire, and when the input is above this threshold, the neuron fires with a current proportional to its input. ReLUs have been found to make it substantially easier to train deep networks (Nair and Hinton, 2010), and are currently very widely used. One disadvantage of ReLUs is that they can wind up in a state in which they are inactive for almost all inputs, meaning that no gradients flow backward through the unit. This, in turn, means that the unit is perpetually stuck in an inactive state, which at a large scale can decrease the network's overall capacity. This is mitigated by using leaky ReLUs, for which even input $< 0$ leads to some activity, allowing for error propagation given any input value.

The softmax function is generally only used at the final layer in classification problems, as it yields a probability distribution based on its input.

### 2.3.3 Learning

Learning in an FFNN happens in two phases. First, in the forward propagation pass, the network sends a given input through the network, and produces some output. Then, the *error* of this output is calculated, as compared to some target label, and this error is sent back through the network, updating the weights of the network so as to make a more accurate prediction given the input in the next

forward pass.[10]

We have already seen the largest part of the forward pass, as in the examples with the AND and XOR functions in Section 2.3. The only remaining part of the forward pass, is how the error of the network is calculated – for this, a loss function is necessary.

#### Loss functions

The loss functions used in NNs, generally fall into two classes – those used for classification problems (i.e. when attempting to predict some discrete class label, out of a finite set of labels), and those used for regression problems (i.e. when attempting to predict some continuous score). Classification is one of the most common cases in NLP (e.g. in POS tagging, NER, language identification, and so on). In such cases, the activation function of the final layer is the softmax function, which allows for interpreting the layer's activations as a probability distribution over the labels under consideration. Most often, the cross-entropy between this predicted probability distribution and the target probability distribution is used to calculate the error, or loss $L$, such that

$$L_{cross-entropy}(\vec{\hat{y}}, \vec{y}) = -\sum_i \vec{y}_i \log \vec{\hat{y}}_i, \tag{2.4}$$

where $L$ denotes the loss function, $\vec{y}$ is the target probability distribution over labels, $\vec{\hat{y}}$ is the model's predicted model distribution given an input $x$. A high error thus indicates that the predicted probability distribution is not consistent with the target probability distribution, and therefore changes should be made accordingly in the backward propagation pass.

[10] This is referred to as *backward propagation of errors*, and is covered later in this section.

Another loss function, common in regression, is the squared error function, defined as

$$L_{squared} = (\hat{y} - y)^2, \tag{2.5}$$

where $\hat{y}$ is the predicted label, and $y$ is the true label. This function is commonly used in regression, and is especially handy for explaining backpropagation, as in the next section.

### Backpropagation

Backward propagation of errors, or *backprop* (Rumelhart et al., 1985; LeCun et al., 1998b), is an algorithm for calculating the gradient of the loss function, for each weight. The gradient can, in turn, be used to update the weights by using an optimisation algorithm, such as gradient descent (discussed further in Section 2.3.3). Intuitively seen, gradient-based methods operate by viewing the errors as a geometric area, and use the slope of the area in which they are (i.e., the gradient) in order to shift weights towards obtaining an error in a minimum of this area, as in Figure 2.5.

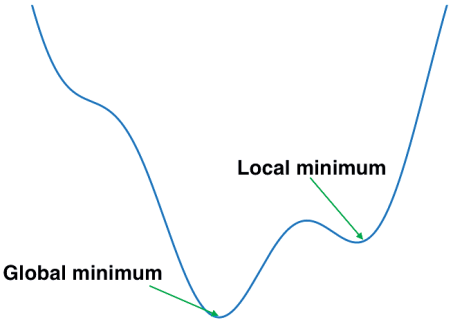


Figure 2.5: Non-convex error surface.

Backprop relies on the fact that the partial derivative of the error of a certain weight $\mathbb{W}_i^j$, with respect to the loss function, can be eas-

ily calculated if we know the partial derivative of the outputs in the layer following that weight. It turns out that this is, indeed, the case, as the derivative of the output layer is quite easily obtained. The output error of a given unit, $\delta_i$, is calculated as

$$\delta_i = \begin{cases} \alpha_i(1-\alpha_i)(\alpha_i - y_i) & \text{for output units } i, \\ \alpha_i(1-\alpha_i)(\alpha_i \sum_{\ell \in L} \delta_\ell \mathbb{W}_{i\ell}) & \text{for other units } i, \end{cases} \tag{2.6}$$

where $\alpha_i$ is the activation of the current unit, $y_i$ is the target output, $L$ is the collection of all units receiving input from the current unit, and $\mathbb{W}_{i\ell}$ is the weight from the current unit to unit $\ell$. Let us consider a concrete example, and go through the forward pass, calculation of the error, and the backward pass. The network in Figure 2.6 shows a neural network with its weights.

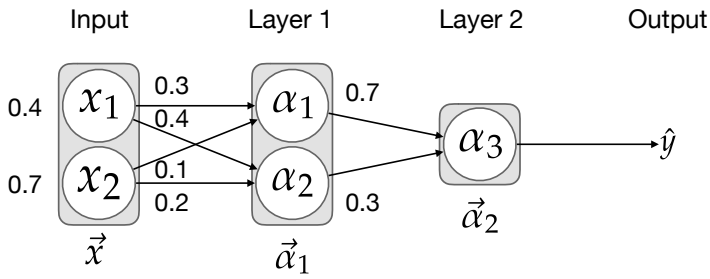


Figure 2.6: A neural network with weights for our backpropagation example.

Assuming that the activation function used is the logistic function, the following calculations hold:

$$\begin{aligned} \alpha_1 &= \sigma(x_1 \times 0.3 + x_2 \times 0.1) = \sigma(0.4 \times 0.3 + 0.7 \times 0.1) = 0.547, \\ \alpha_2 &= \sigma(x_1 \times 0.4 + x_2 \times 0.2) = \sigma(0.4 \times 0.4 + 0.7 \times 0.2) = 0.574, \\ \alpha_3 &= \sigma(\alpha_1 \times 0.7 + \alpha_2 \times 0.3) = \sigma(0.19 \times 0.7 + 0.3 \times 0.3) = 0.635. \end{aligned} \tag{2.7}$$

Assuming that the target output is $y = 0.4$, we can now calculate the error. Applying equation 2.6, we can obtain the error of the output,

namely

$$\begin{aligned}\delta_3 &= \alpha_3(1-\alpha_3)(\alpha_3 - y),\\ &= 0.635(1-0.635)(0.635-0.4),\\ &= 0.054.\end{aligned} \tag{2.8}$$

The errors of the two hidden units can also be calculated, yielding

$$\begin{aligned}\delta_2 &= \alpha_2(1-\alpha_2)(\alpha_2\delta_3\mathbb{W}_{2\ell}),\\ &= 0.547(1-0.547)(0.547\times 0.027\times 0.3),\\ &= 0.001,\end{aligned} \tag{2.9}$$

and

$$\begin{aligned}\delta_1 &= \alpha_1(1-\alpha_1)(\alpha_1\delta_3\mathbb{W}_{1\ell}),\\ &= 0.547(1-0.547)(0.547*0.027*0.7),\\ &= 0.002.\end{aligned} \tag{2.10}$$

We now need to update the weights used, via gradient descent. This can be done by shifting the weights with some constant with respect to the error obtained,

$$\Delta\mathbb{W}_{ij} = -\gamma\alpha_i\delta_j, \tag{2.11}$$

where $\Delta\mathbb{W}_{ij}$ is the amount with which to change $\mathbb{W}_{ij}$ (i.e., the weight between the firing and receiving unit), $\gamma$ is some learning rate, $\alpha_i$ is the activation of the firing unit, and and $\delta_j$ is the error of the receiving unit. Hence, if we set $\gamma = 1$, the changes of the weights are calculated as

$$\begin{aligned}\Delta\mathbb{W}_{x_1,\alpha_1} &= -\gamma x_1\delta_1 = -1\times 0.4\times 0.002 = -0.0008,\\ \Delta\mathbb{W}_{x_1,\alpha_2} &= -\gamma x_1\delta_2 = -1\times 0.4\times 0.001 = -0.0004,\\ \Delta\mathbb{W}_{x_2,\alpha_1} &= -\gamma x_2\delta_1 = -1\times 0.7\times 0.002 = -0.0014,\\ \Delta\mathbb{W}_{x_2,\alpha_2} &= -\gamma x_2\delta_2 = -1\times 0.7\times 0.001 = -0.0007,\\ \Delta\mathbb{W}_{\alpha_1,\alpha_3} &= -\gamma\alpha_1\delta_3 = -1\times 0.547\times 0.054 = -0.030,\\ \Delta\mathbb{W}_{\alpha_2,\alpha_3} &= -\gamma\alpha_2\delta_3 = -1\times 0.574\times 0.054 = -0.031.\end{aligned} \tag{2.12}$$

Using the new weights yields the following activations in the next forward pass, given the same input:

$$\begin{aligned}
\alpha_1 &= \sigma(0.4 \times (0.3 + \Delta\mathbb{W}_{x_1,\alpha_1}) + 0.7 \times (0.1 + \Delta\mathbb{W}_{x_2,\alpha_1}) = 0.547,\\
\alpha_2 &= \sigma(0.4 \times (0.4 + \Delta\mathbb{W}_{x_2,\alpha_1}) + 0.7 \times (0.2 + \Delta\mathbb{W}_{x_2,\alpha_2}) = 0.574,\\
\alpha_3 &= \sigma(0.547 \times (0.7 + \Delta\mathbb{W}_{\alpha_1,\alpha_3}) + 0.574 \times (0.3 + \Delta\mathbb{W}_{\alpha_2,\alpha_3}) = 0.627,
\end{aligned} \tag{2.13}$$

and the output error

$$\begin{aligned}
\delta_3 &= \alpha_3(1-\alpha_3)(\alpha_3 - y),\\
&= 0.318(1-0.318)(0.318-0.4),\\
&= 0.053,
\end{aligned} \tag{2.14}$$

which is smaller than the previous error where $\delta_3 = 0.054$. This process is repeated with other training examples, until some criterion is reached, such as a sufficiently low average loss.

#### Optimisation Methods

Backpropagation, as described in the previous section, can provide us with the derivatives of the error surface. This can be used in a variety of ways to update the weights. What we just saw, in Equation 2.11, is known as gradient descent. One of the most commonly used optimisation methods is Stochastic Gradient Descent (SGD). In SGD, a minibatch of $n$ samples is drawn from the training set, the gradient is calculated based on this batch, and the weights are then updated accordingly (Bottou, 1998). Other algorithms, such as AdaGrad (Duchi et al., 2011) and RMSProp (Hinton, 2012), learn and adapt the learning rate ($\gamma$) for each weight. Modifying the learning rate in this manner can both increase the rate at which the error decreases, and lead to lower overall errors. A recent and increasingly popular optimisation method is Adam, which is similar to RMSProp and yields better results on a many problems (Kingma and Ba, 2014). The choice

of optimisation method is not all that straightforward, and no real consensus exists for how this should be done (Schaul et al., 2014). Hence, commonly, trial-and-error is applied in order to make this choice, by experimentally investigating performance on a development set.

#### Finding the global minimum

The goal of an optimisation algorithm is to find the global minimum, as shown in Figure 2.5. What so-called *gradient-based* optimisation algorithms do, is to calculate the derivative with respect to this error surface, and shift the weights so as to move towards the closest of all local minima. Such local minima can, however, be the source of a host of problems, if the loss is high compared to the global minimum. This is a frequently occurring issue, and it is possible to construct small neural networks in which this scenario appears (Sontag and Sussmann, 1989; Brady et al., 1989; Gori and Tesi, 1992). It turns out, however, that practically speaking, when considering larger neural networks, it is not particularly important to find the global minimum. This has to do with the fact that, in the case of supervised learning with deep neural networks, most local minima appear to have a low loss function value, roughly equivalent to that of the true global minimum (Saxe et al., 2013; Dauphin et al., 2014; Goodfellow et al., 2015; Choromanska et al., 2015).

#### Parameter Initialisation

There are several methods for initialising the weights in a neural network. Naively, one might think to set the all weight matrices $\mathbb{W} = 1$, however due to how backpropagation works, this will result in all hidden units representing the same function, and receiving the exact same weight updates. Therefore, some random process is required. Common methods include those introduced by Glorot and

Bengio (2010), and Saxe et al. (2013). When employing the ReLU activation function, He et al. (2015b) show that weights should be initialised based on a Gaussian distribution with standard deviation $\sqrt{\frac{2}{d_{in}}}$, where $d_{in}$ is the input dimensionality. In the case of recurrent neural networks, which are covered in Section 2.4, particular care needs to be taken, and weight initialisation is often done using orthogonal matrices (cf. Goodfellow et al.[p.404], 2016).

#### Regularisation in Neural Networks

One of the most common problems when training an ML system in general, is that of overfitting – and neural networks are no exception. Overfitting occurs when the network does not generalise to data outside of the training set, while having a low loss on the training set itself. Generalisation is one of the most important parts of learning, as learning without generalisation is simply the memorisation of a training set. A model which has only memorised the training set is of little practical value, as it will most likely fail miserably on unseen examples. In order to avoid overfitting, regularisation techniques are typically employed. The probably most common regularisation technique used today, is dropout (Srivastava et al., 2014). In dropout, every activation has a probability $p$ of not being included in the forward and backward passes, during training. This procedure leads to significantly lower generalisation error, as the network needs to be more robust, and less reliant on specific units. In the case of recurrent neural networks, which are covered next, specific variants of dropout exist. such as recurrent dropout (Semeniuta et al., 2016), or variational dropout (Gal and Ghahramani, 2016) in which the same dropout mask is used for each time step. Another commonly used manner of regularisation is *weight decay*, in which the magnitudes of weights are decreased according to some criterion (Krogh and Hertz, 1992).

## 2.4 Recurrent Neural Networks

Although FFNNs are suitable for many problems, they do not take the structure of the input into account. Although it is possible to attempt to enforce this in such a network, this has several disadvantages, such as the fact that the amount of parameters which need to be tuned can become prohibitively large. Luckily, there are architectures for dealing with structure, as this is not entirely unimportant when considering natural language. Two such approaches are covered in the following sections.

Recurrent Neural Networks (RNNs) are an extension of feed-forward neural networks, which are designed for sequential data (Elman, 1990). They can be thought of as a sequence of copies of the same FFNN, each with a connection to the following time step in the sequence, sharing parameters between time steps. RNNs take a sequence of *arbitrary length* as input ($x_1, x_2, \ldots, x_t$), and return another sequence ($\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_t$). Each $x_t$ in the input sequence is a vector representation of element $n_t$ in the sequence. Each $\hat{y}_t$ in the output sequence can take advantage of information in the sequence up to step $t$ in the input sequence. This is illustrated in Figure 2.7. Each layer is shown as containing only one unit, which is here meant as an abstraction depicting the entire internal representation of the RNN. The left side of the figure depicts an FFNN with a loop, whereas the right side shows the *unrolled* version of the network. The output of the hidden layer is passed as an input to the hidden layer in the next time step.

An RNN is essentially a group of FFNNs with connections to one another. This connection is a sort of loop, going from the hidden layer of the network at time $x_t$ to the hidden layer at $x_{t+1}$. In other words, $\vec{\hat{y}_t}$ is calculated as

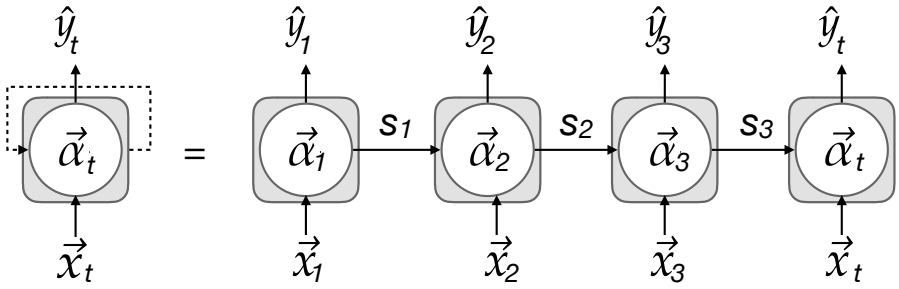


Figure 2.7: A simple RNN with a connection from the hidden state of the previous time step to the current side step. Left side shows the FFNN with the loop, whereas the right side shows the unrolled network.

$$\begin{aligned} \vec{z}_t &= \mathbb{W}_s \vec{x}_t + \mathbb{U}\vec{s}_{t-1}, \\ \vec{s}_t &= \sigma_s(\vec{z}_t), \\ \vec{\hat{y}}_t &= \sigma_y(\mathbb{W}_y \vec{s}_t), \end{aligned} \tag{2.15}$$

where $\mathbb{W}_s$ is the matrix of weights for the current time step's input ($\vec{x}_t$), $\mathbb{U}$ is a weight matrix for the connections from the previous time step, $\vec{s}_t$ is a state vector representing the history of the sequence, $t$ is the index of the current time step, $\mathbb{W}_y$ is the matrix of weights for the output, and the rest is defined as for FFNNs. This is what is also referred to as an Elman net, or a Simple RNN (Elman, 1990). The advantage of having access to $\vec{s}$, is that the network can take advantage of preceding information when outputting $\vec{\hat{y}}_t$. For instance, in the case of POS tagging, if the current input is *fly*, and the state vector shows that the previous word was *to*, we most likely want to output the tag *verb*. Hence, in this way, the prediction at each time step is conditioned on the inputs in the entire preceding sequence. There are also variants of this, in which the net's outputs are used to cal-

culate the state vector, as in the case of Jordan nets (Jordan, 1997), which is defined such that

$$\vec{z}_t = \mathbb{W}_s \vec{x}_t + \mathbb{U}\vec{\hat{y}}_{t-1}. \tag{2.16}$$

Although RNNs, in theory, can learn long dependencies (i.e. that an output at a certain time step is dependant on the state at a time step far back in the history), and can handle input sequences of arbitrary length, they are in practice heavily biased to the most recent items in the given sequence, and thus difficult to train on long sequences with long dependencies (Bengio et al., 1994). For instance, in the case of language modelling, given a sentence such as *My mother is from Finland, so I speak fluent* ..., it is quite likely that the omitted word should be *Finnish*. However, as the distance between such dependencies grows, it becomes increasingly difficult for an RNN to make use of such contextual information. In general, this is because deep neural networks suffer from having *unstable* gradients, as the gradients calculated by backprop (Section 2.3.3) are dependant on the output of the network, which can be quite far away from the first layers in the network. One problem with this is that this can lead to *vanishing* gradients (i.e. the gradient becomes very small). This happens since the gradient in early layers of the network are the result of a large number of multiplication operators on numbers $< 1$. One might consider the fact that, since these multiplications involve the weights in the network, we might just set the weights to be really large. Although this might seem like a good idea, this will likely lead to the converse of the issue one is trying to avoid, namely that of *exploding* gradients. Since most common optimisation methods are gradient-based, this is problematic (see Section 2.3.3 for optimisation details).

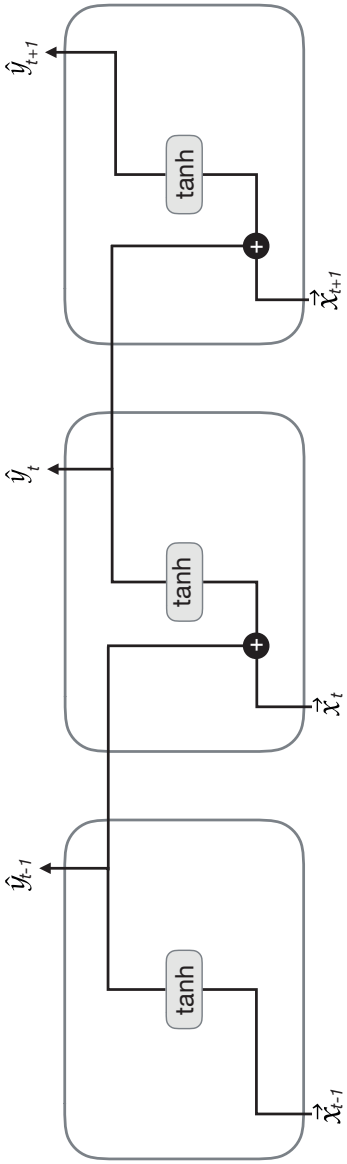


Figure 2.8: Internal view of an RNN in three time steps.

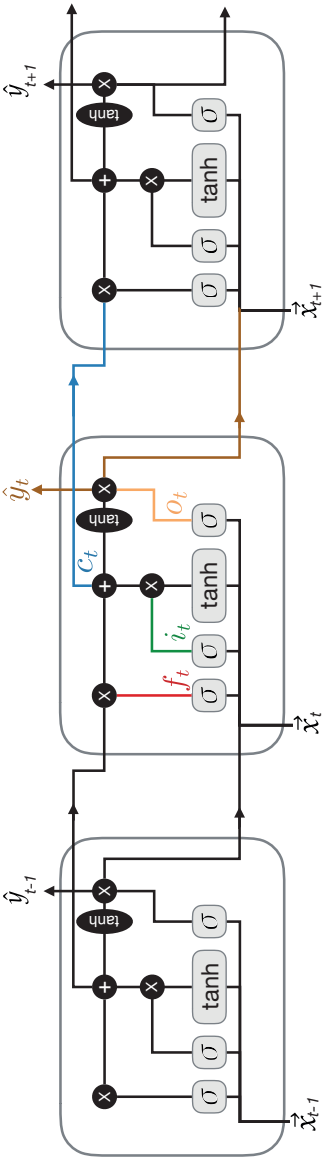


Figure 2.9: Internal view of an LSTM in three time steps.

### 2.4.1 Long Short-Term Memory

Previous work has attempted to solve this problem by adapting the optimisation method used (Bengio et al., 2013; Pascanu et al., 2013; Sutskever et al., 2014), however the more successful approach has been to modify the neural network architecture itself. Because of such efforts, there are several types of RNNs specifically designed to cope with this issue, essentially by enforcing a type of protection of the memory of the history of the input sequence, storing and maintaining important features, while neglecting and forgetting unimportant features. One such method, namely Long Short-Term Memory (LSTM), was described in Hochreiter and Schmidhuber (1997), and saw an explosion in popularity around 2014, following several influential papers (e.g. Sundermeyer et al. (2012); Sutskever et al. (2014); Dyer et al. (2015)). An LSTM is an extension of RNNs, with *memory cells*, engineered to cope with the issue of unstable gradients, and have been shown to be able to capture long-range dependencies (Hochreiter and Schmidhuber, 1997; Cho, 2015). For an overview of the many variations of LSTMs which appear in the literature, see Greff et al. (2016).

Whereas an RNN only has a single internal layer (Figure 2.8), typically with a *tanh* (hyperbolic tangent) activation, an LSTM is somewhat more complicated (Figure 2.9). An LSTM contains gates, denoted in the figure by the $\sigma$ layers, which are used to modify the extent to which old information is remembered or forgotten. Part of the explanation for LSTMs involves the observation that they, on the surface, can be seen as a combination of Elman nets and Jordan nets, in that both the cell state and the hypothesis are passed between

states. In detail, an LSTM is implemented as follows

$$
\begin{aligned}
f_t &= \sigma(\mathbb{W}_f x_t + \mathbb{U}_f \hat{y}_{t-1} + b_f), \\
i_t &= \sigma(\mathbb{W}_i x_t + \mathbb{U}_i \hat{y}_{t-1} + b_i), \\
o_t &= \sigma(\mathbb{W}_o x_t + \mathbb{U}_o \hat{y}_{t-1} + b_o), \\
c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(\mathbb{W}_c x_t + \mathbb{U}_c \hat{y}_{t-1} + b_c), \\
\hat{y}_t &= o_t \circ \sigma(c_t),
\end{aligned}
\tag{2.17}
$$

where $f_t$ represents the output of the *forget gate*, $i_t$ represents the output of the *input gate*, $o_t$ represents the output of the *output gate*, $c_t$ represents the *cell state*, $\hat{y}_t$ represents the output vector, $\mathbb{W}$ and $\mathbb{U}$ represent the weight matrices, $x_t$ represents the current input, and $b$ represents bias units. Each of these parts are covered in detail in the following sections.

### Cell state

The cell state, $c_t$, is the line coded in blue in Figure 2.9. It is similar to the state vector, $\vec{s}$, in a simple RNN, but its content is maintained and protected by three gates. The forget gate's output $f_t$ determines to which extent each feature in the cell state should be kept. This is done by observing the current input $x_t$, and the previous time step's output $\hat{y}_{t-1}$.

For instance, say we have a POS tagger which at time $t$ observes a word which could be either a noun or a verb (e.g. *fly*). If the previous word was *to*, this would be useful to keep in mind in order to better predict the next tag. Following this time step, however, we might want to forget about this word, when predicting the next tag.

Adding information to the cell state happens in two steps. We first decide on which values in the cell state to update again observing $x_t$ and $\hat{y}_{t-1}$, again deciding for each dimension the extent to which we will add information. This is denoted by the input gate $i_t$. The vector which is added to the cell state, is calculated by passing $x_t$ and $\hat{y}_{t-1}$

through a non-linearity, marked by tanh in the figure. In the POS tagging example, we want to add the information regarding the preceding determiner. These two vectors are then summed, resulting in our new cell state $c_t$.

#### Output

Finally, we need to output a value from the current time step. This output is based on the cell state, $c_t$, which is first run through a non-linearity (usually tanh), and filtered again by $o_t$, which also observes $x_t$ and $\hat{y}_{t-1}$, when deciding on which dimensions to keep, and to what extent.

#### Gated Recurrent Units

In addition to the many LSTM variants (Greff et al., 2016), Gated Recurrent Units (GRUs) represent a different variant of gated RNNs which was independently developed to LSTMs and similar in both purpose and implementation (Cho et al., 2014). The main difference between LSTMs and GRUs is the fact that GRUs do not have separate memory cells, and only include two gates – an update gate, and a reset gate (Chung et al., 2014). This in turn means that GRUs are computationally somewhat more efficient than LSTMs. In practice, both LSTMs and GRUs have been found to yield comparable results (Chung et al., 2014; Jozefowicz et al., 2015). On a general level, the performance of various gated RNN architectures is, at least in the case of large amounts of data, closely tied to the number of parameters (Collins et al., 2017; Melis et al., 2017).

#### Bi-directionality

Many properties of language depend on both preceding and proceeding contexts, so it is useful to have knowledge of both of these contexts simultaneously. This can be done by using a bi-directional RNN

variant, which makes both forward and backward passes over sequences, allowing it to use both contexts simultaneously for the task at hand (Schuster and Paliwal, 1997; Graves and Schmidhuber, 2005; Goldberg, 2015). Bi-directional GRUs and LSTMs have been shown to yield high performance on several NLP tasks, such as POS tagging, named entity tagging, and chunking (Wang et al., 2015; Yang et al., 2016; Plank et al., 2016).

### 2.4.2 Common use-cases of RNNs in NLP

In NLP, there are four general scenarios for producing some sort of analysis for a given text. Consider that we have the following sentence as input:

(2.18) *I'm not fussy.*[11]

We might want to analyse this unit as a whole, for instance in order to judge that the text is in English, and not in some other language, or to determine the native language of the person writing it, or the sentiment of the text itself. This can be referred to as a **many-to-one** scenario, since we have several smaller units (e.g. words or characters), which we want to translate into a single score or class, depending on the task at hand.

On the other hand, we might want to analyse the sentence word by word, by assigning, e.g., a part-of-speech (POS) tag or a semantic tag to each word in the sentence. This can be referred to as a **one-to-one** scenario, since every single unit in the text (e.g. each word token) has a direct correspondence to a single tag.[12]

[11] PMB 76/2032, Original source: Tatoeba

[12] The term 'one-to-one' is also used for simple classification cases where there is no sentential context available. We see this as simply being a special case in which the sequence length $= 1$. This is equivalent to the relation between FFNNs and RNNs, in which an FFNN can be seen as a special case of RNNs (or vice versa).

We might also want to carry out some task in which the sentence should be translated to some other form, for instance translating the sentence to German, or some other language. If the sentence was written in some non-standard form of English, we might want to produce a normalised version of the sentence, or in a different setting we might want to generate an inflected form of some word in the same language. This can be referred to as a **many-to-many** scenario, as there is no structural one-to-one correspondence between the input $X$ and the output $Y$.

A final logically possible case, is the **one-to-many** scenario. This is a highly uncommon scenario, as it is not generally the case that one tries to predict several things from an atomic unit. Although one could argue that some tasks fit this scenario, such as caption generation, this is not really a one-to-many scenario, as the image is not an atomic unit, but is read by the NN as a matrix of pixels.

A schematic overview of the three relevant scenarios is given in Figure 2.10. The versatility of these three scenarios is evident when observing the current NLP scene, in which common practise is to cast a problem to fit one of these scenarios, and to then throw a Bi-LSTM at the problem.

#### Many-to-one

Many NLP tasks deal with going from several smaller units to a single prediction. This essentially means that these units need to be compressed into a single vector, onto which a softmax layer can be applied in order to arrive at a probability distribution over the classes at hand (e.g. a set of languages to identify). For this type of problem, a number of possibilities exist, such as

1. Averaging the vectors representing each unit in the sentence (i.e. average pooling);

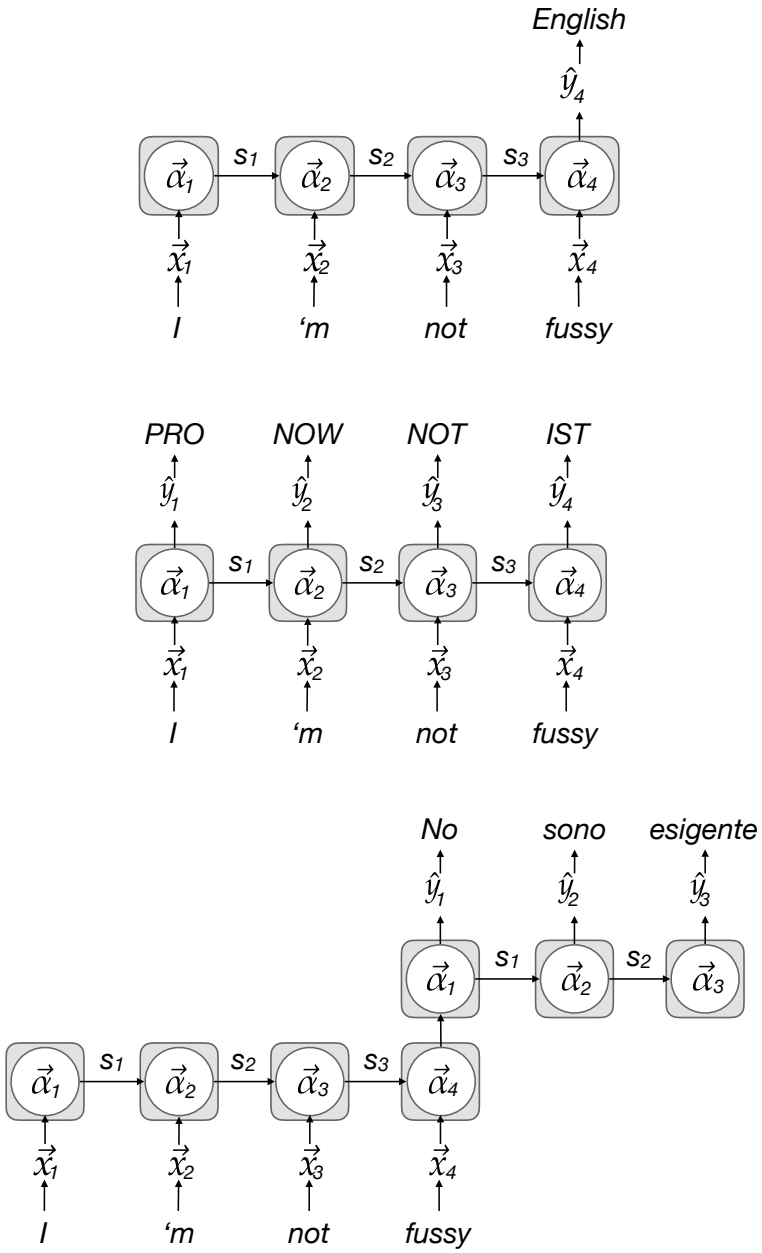


Figure 2.10: Common scenarios in which RNNs are applied in NLP. From top to bottom: many-to-one, one-to-one, and many-to-many.

2. Applying an RNN and using the final state output vector as a representation of the sentence (depicted in Figure 2.10);

3. Applying convolutions in order to arrive at a condensed representation.[13]

Approach 1) is the most simplistic of these, and has been successfully applied in previous work (e.g. Socher et al. (2013a); Zhang et al. (2015a)). The main advantage of this approach is indeed its simplicity, as calculating the mean of the vectorial representations of a sentence is both a very cheap operation, as well as an operation which allows for the application of a simple FFNN on top of this vector. With this approach, however, the structure inherent in the natural language signal is left unexploited.

Approaches 2) and 3) both offer more expressive power as compared to approach 1), as they both take advantage of the structure inherent in the input signal. This is not the case when summing the vectors, which is in a way analogous to a bag-of-words approach. Structure is naturally of paramount importance in natural language, so taking advantage of this is good. Although structure in natural language is generally hierarchical, even using the sequential structure is better than assuming no structure at all. There are, however, some recent architectures which do encode the hierarchical structure of language, such as tree LSTMs (Tai et al., 2015), and RNN-Grammars (Dyer et al., 2016).

These three approaches are meant to give an overview of some straight-forward manners of obtaining such representations. Other, more sophisticated approaches, include skip-thought vectors, in which sentence-level representations are learned with the objective of being able to predict surrounding sentences in a document (Kiros et al., 2015). A systematic overview of such methods is given by Hill et al.

[13] Convolutional Neural Networks are described in Section 2.5.

(2016), who conclude that the best suited approach depends on the intended application of such representations.

Another use case for this approach is when building hierarchical models, in the sense that one, e.g., may want to have word representations which are aware of what is going on on a sub-word level. For this, one might apply a many-to-one RNN, and use the final state output vector as a word vector (Ballesteros et al., 2015; Plank et al., 2016). Alternatively, one can use convolutions to arrive at this type of word vector, as in dos Santos and Zadrozny (2014).

#### One-to-one

The one-to-one case is perhaps one of the most common scenarios in NLP. This covers tagging task scenarios, as well as simple classification scenarios. Many NLP tagging tasks have seen relatively large improvements when applying variants of RNNs similarly to what is depicted in Figure 2.10. Recently, gated variants such as LSTMs and GRUs, often in a bi-directional incarnation are applied (Wang et al., 2015; Huang et al., 2015; Yang et al., 2016; Plank et al., 2016). Such RNNs are highly suited for this type of task, as it is highly informative for, e.g., POS tagging to know which words occur both before and after the word at hand. Such dependencies might also have quite large spans, which both LSTMs and GRUs are able to capture well. To contrast with older feature-based models, it would require a fair bit of feature engineering to decide on what types of spans to include in feature representations, lest one wishes to suffer from the sparsity of simply using $n$-gram features with large values of $n$.

#### Many-to-many

This paradigm is also what is frequently referred to as an encoder-decoder architecture in the literature, or sequence-to-sequence learning problems (Bahdanau et al., 2015; Sutskever et al., 2014). A fre-

quent approach here, for instance in machine translation, is to apply an RNN from which one takes the final time step's output to be a representation of the entire sentence, as represented in Figure 2.10, which may seem like a bold thing to do.[14] It turns out that this is in fact often not sufficient, although one can obtain surprisingly good translations this way. However, results improve dramatically when going a step further by incorporating an attentional mechanisms. This is not focussed upon in this thesis, and will not be explained in full detail. Essentially, an attention mechanism can learn which parts of the source sentence to attend to, when producing the target sentence translation. For instance, such a mechanism might learn an implicit weighted word-alignment between the source and target sentences, thus facilitating translation.

Many NLP tasks can be solved with a many-to-many approach. Machine translation has already been mentioned, and has in no small degree been the driving force behind research in this direction. Apart from this, the approach has been applied to morphological inflection (Kann and Schütze, 2016; Cotterell et al., 2016; Östling and Bjerva, 2017; Cotterell et al., 2017), AMR parsing (Barzdins and Gosko, 2016; Konstas et al., 2017; van Noord and Bos, 2017b,a), language modelling (e.g. Vinyals et al., 2015), generation of Chinese poetry (Yi et al., 2016), historical text normalisation (Korchagina, 2017), and a whole host of other tasks.

## 2.5 Convolutional Neural Networks

Certain machine learning problems, such as image recognition, deal with input data in which spatial relationships are of utmost importance. While simpler image recognition problems, such as handwrit-

[14] *You can't cram the meaning of a whole sentence into a single vector!*
–Ray Mooney, as communicated by Kyunghyun Cho in his NoDaLiDa 2017 keynote (`https://play.gu.se/media/1_xt08m5je`)

ten digit recognition, can be carried out relatively successfully with simple FFNNs, this is often not sufficient. Recall that the input for an FFNN is simply a single vector $\vec{x}$, meaning that the network has no notion of adjacency between, e.g., two pixels. A Convolutional Neural Network (CNN) is a type of network explicitly designed to take advantage of the spatial structure of its input. The origins of CNNs go back to the 1970s, but the seminal paper for modern CNNs is considered to be LeCun et al. (1998a), although other work exists in the same direction (e.g., LeCun et al. (1989); Waibel et al. (1989)). CNNs have been used extensively in NLP, and can in many cases be used instead of an RNN (contrast, e.g., dos Santos and Zadrozny (2014) who use CNNs for character-based word representations, and Plank et al. (2016) who use RNNs for the same purpose).

Although NLP is the focus of this thesis, we will approach CNNs from an image recognition perspective, as this is somewhat more intuitive. This is in part due to the fact that image recognition was the intended application of CNNs upon their conception. On a general level, convolutions can be carried out on input of arbitrary dimensionality. As mentioned, two-dimensional input (e.g. images) were the original target for CNNs. More recent work has extended this to three-dimensional input (e.g. videos). In the case of NLP, it is often the case that one-dimensional input is used, for instance applying a CNN to a text string. There are three basic notions which CNNs rely upon: local receptive fields, weight sharing, and pooling.

### 2.5.1 Local receptive fields

In the case of image recognition, an image of $n \times n$ pixels can be coded as an input layer of $n \times n$ units.[15] In a CNN, this input is processed by sliding a window of size $m \times m$ across this image. This window, or patch, is known as a local receptive field. After passing this window

[15] This is assuming greyscale, i.e., one value per pixel.

over the input image, the following layer contains a representation based on $m \times m$ sized slices of the input image. Intuitively, this can be seen as blurring the input image somewhat, as the spatial dimensions of the image are generally reduced through this process.

The length with which this window moves is referred to as its *stride*, and is most often set to $1$, meaning that the window simply shifts by one pixel at a time. Although stride lengths of 2 and 3 are encountered in the literature, it is fairly uncommon to see larger stride lengths than this. Figure 2.11 shows a convolution, where $n = 4$, $m = 2$, a stride of $2$ is used, which yields a new layer with size $2 \times 2$.

### 2.5.2 Weight sharing

A key notion of CNNs is the fact that weights are shared between each such local receptive field, and the units to which they are attached. Hence, in our example, rather than having to learn $n \times i = 64$ weights (where $i = n \times n$ is the total number of units in the first hidden layer), as in an FFNN, only $m \times m = 4$ parameters need to be learned. Therefore, all units in the first hidden layer capture the same type of features from the input image in various locations. For instance, imagine you want to identify whether a picture contains cats (as in Figure 2.12). In this figure, units in the first hidden layer might encode some sort of *cat detector*. The fact that weights are shared in this manner, results in CNNs being robust to translation invariance. This means that a feature is free to occur in different regions in the input image. Intuitively, taking our cat detector as an example, it is naturally the case that a cat is a cat, regardless of where in the image it happens to hide.

Each such *cat detector*, is referred to as a *feature map*, or a *channel*.[16] For a CNN to be useful, normally more than one feature map

[16]The *channel* terminology makes sense when considering an input image in, e.g., RGB formatting, in which the intensities of each colour is represented in a separate colour channel.

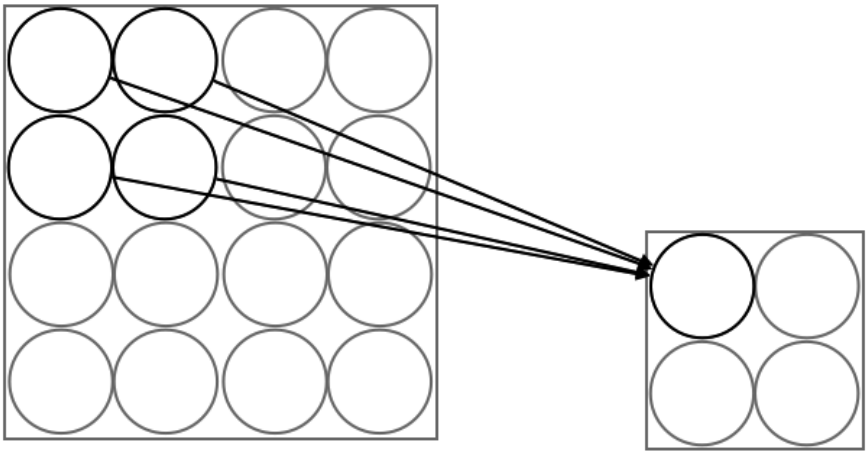

Figure 2.11: Illustration of a local receptive field of size $2x2$, which results in a new image of size $2x2$ due to the stride length being $2$.

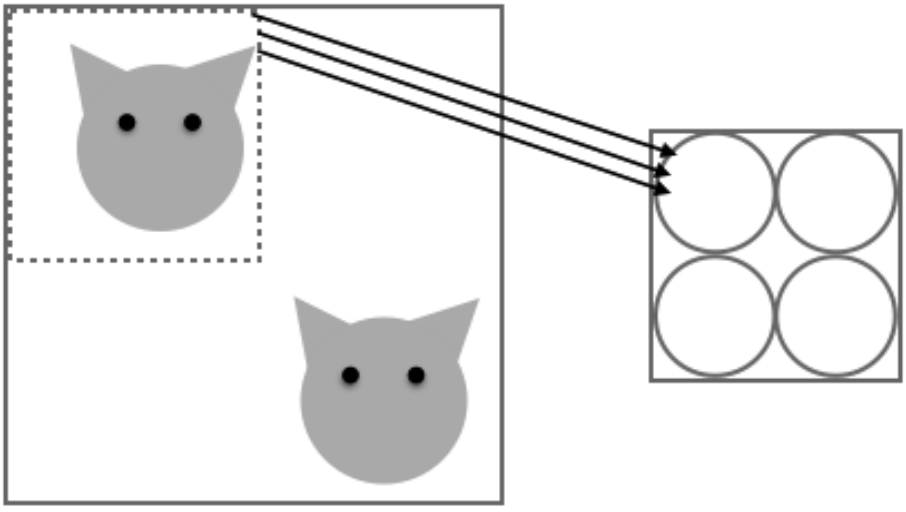

Figure 2.12: Weight sharing example with a cat. The dotted line represents the local receptive field used, the first square represents the entire input domain, and the second square represents the following convolutional layer.

is learnt. That is to say, while the figures shown so far only show a single feature map, an input image is normally mapped to several smaller images. As an example, another feature map in Figure 2.12 might learn a *dog detector*. A more realistic example can be found in facial recognition, in which one feature map might learn to detect eyes, while another learns to detect ears, and yet another learns to detect mouths. Local receptive fields generally speaking look at all feature maps of the previous layers, hence the combination of eyes, ears, and mouths might be used to learn a feature map representing an entire face.

The fact that we generally map to several feature maps means that, at each layer, the spatial size of the image shrinks (i.e. $m < n$), while the depth of the image increases, as shown in Figure 2.13. Finally, following a series of convolutional layers, it is common practise to attach an FFNN prior to outputting predictions.

In NLP, the situation is somewhat different, as we normally do not have an image as input, but rather some sort of textual representation. Commonly, this will either be a string of words, characters, or bytes. An intuition for how this works, is that something resembling an n-gram feature detector is learnt given a window. This type of approach has been applied successfully in various tasks, for instance to obtain word-level representations which take advantage of sub-word information (dos Santos and Zadrozny, 2014; Bjerva et al., 2016b), and for sentence classification (Kim, 2014).

### 2.5.3 Pooling

A pooling layer takes a feature map and condenses this into a smaller feature map. Each unit in a pooling layer summarises a region in the previous layer, generally using a simple arithmetic operation. Frequently, operations like maximum pooling (max pooling) or average pooling are used (Zhou and Chellappa, 1988). These operations take, e.g., the maximum of some region to be a representation of that en-

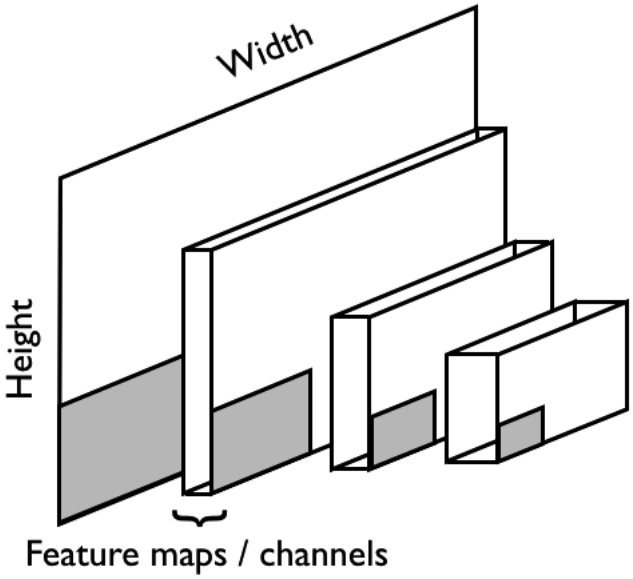


Figure 2.13: General CNN structure. Each layer shrinks the width and height of the input image, and increases the number of feature maps. The grey regions denote the sizes of the local receptive fields.

tire region, thus reducing dimensionality by essentially applying simple non-linear downsampling. Max pooling can thus be seen as a way for each max pooling unit to encode whether or not a feature from the previous layer was found anywhere in the region which the unit covers. The intuition is that the downsampled version of the feature map, which yields feature locations which are rough, rather than precise, is sufficient in combination with the relative location to other such downsampled features. Importantly, this operation reduces the dimensionality of feature maps, thus reducing the number of parameters needed in later layers.

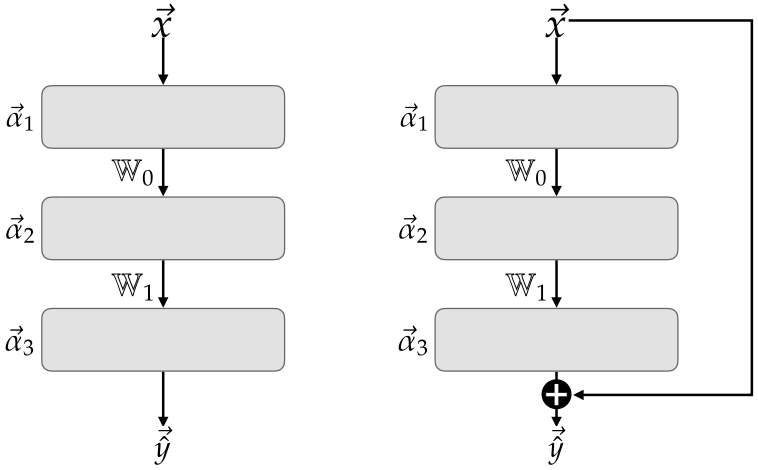


Figure 2.14: Illustration of a residual network (right) as compared to a standard network without skip connections (left). The skip connection here passes the input vector $\vec{x}$ and adds this to the activations $\vec{\alpha}_3$, thus providing the network with a shortcut. The squares represent an abstract block of weights, such as, e.g., a fully connected layer in an FFNN, a convolutional block in a CNN, or an LSTM cell.

## 2.6 Residual Networks

Residual Networks (ResNets) define a special class of networks with skip connections between layers, as depicted in Figure 2.14. This facilitates the training of deeper networks, as these skip connections ease the propagation of errors back to earlier layers in the network.

Such skip connections are referred to as residual connections, and can be expressed as

$$\begin{aligned} y_l &= h(x_l) + \mathcal{F}(x_l, \mathcal{W}_l), \\ x_{l+1} &= f(y_l), \end{aligned} \tag{2.19}$$

where $x_l$ and $x_{l+1}$ are the input and output of the $l$-th layer, $\mathcal{W}_l$ is the

weights for the $l$-th layer, and $\mathcal{F}$ is a residual function (He et al., 2016) such as the identity function (He et al., 2015a), which we also use in our experiments. Although ResNets were developed for the use in CNNs, the skip connections are currently being used, e.g., in LSTMs (Wu et al., 2016). ResNets can be intuitively understood by thinking of residual functions as paths through which information can propagate easily. This means that, in every layer, a ResNet learns more complex feature combinations, which it combines with the shallower representation from the previous layer. This architecture allows for the construction of much deeper networks. ResNets have recently been found to yield impressive performance in image recognition tasks, with networks as deep as 1001 layers (He et al., 2015a, 2016), and are thus an interesting and effective alternative to simply stacking layers. Another useful feature of ResNets is that they act as ensembles of relatively shallow networks, which may help to explain why they are relatively robust to overfitting in spite of their large number of parameters (Veit et al., 2016).

ResNets have recently been applied in NLP to morphological reinflection (Östling, 2016), language identification (Bjerva, 2016), sentiment analysis and text categorisation (Conneau et al., 2016), semantic tagging (Bjerva et al., 2016b), as well as machine translation (Wu et al., 2016). Recently proposed variants include wide residual networks, with relatively wide convolutional blocks, showing that resnets do not necessarily need to be as deep as the 1001 layers used in previous work (Zagoruyko and Komodakis, 2016).

## 2.7 Neural Networks and the Human Brain

While there are numerous reasons to use neural networks, there are camps which might argue for applying NNs because they are *'biologically motivated'*. While it can be tempting to use the conceptual metaphor (Lakoff and Johnson, 1980) of NEURAL NETWORKS ARE THE

BRAIN, this can be rather misleading. While usage of misleading metaphors can seem innocent, it has been shown that they do affect reasoning (Thibodeau and Boroditsky, 2013). Therefore, a misleading metaphor should certainly not be used as an argument for the usage of neural networks – there are plenty of other reasons for that.

Now, you might ask, is this metaphor really as misleading as it seems? At best, neural networks (as used in NLP) are a mere caricature of the human brain. On a physiological level, there is evidence that neurons do more than simply outputting some activation value based on a weighting of its inputs. For instance, recent research has shown that a single neuron can encode temporal response patterns, without relying on temporal information in input signals. Hence, the nature of how neurons work is quite different from what is encoded in a neural network, for instance in terms of information storage capacity (Jirenhed et al., 2017). There is in fact compelling evidence that memory is not coded in (sets of) synapses, but rather internally in neurons (cf. Gallistel and King, 2011, and Gallistel, 2016 and references therein, notably Johansson et al. (2014)). Additionally, backpropagation is not biologically plausible, although there is work on making biologically plausible neural networks (Bengio et al., 2015).

#### Convolutional Neural Networks and the Brain

Similarly to the ReLUs discussed in Section 2.3.2, CNNs are biologically inspired. When CNNs were invented (LeCun et al., 1998a), this was inspired by work which proposed an explanation for how the world is visually perceived by mammals (Hubel and Wiesel, 1959, 1962, 1968). The attempts to reverse-engineer a similar mechanism, as in CNNs, have proven fruitful, as CNNs are indeed highly suitable for image recognition. Furthermore, recent research has found some correlations between the representations used by a CNN, and those encoded in the brain in a study where the CNN could identify which image a human participant was looking at roughly 20% of the time

(Seeliger et al., 2017). However, it remains to be seen whether anything similar to this is even plausible for natural language.

## 2.8 Summary

In this chapter, an intuitive and theoretically supported overview of neural networks was given, including a practical overview NLP. We have seen that many common NLP problems can be classified into three categories: one-to-one, one-to-many, and many-to-many. Appropriate deep learning architectures suitable for each of these categories were suggested.

While this chapter is meant to be a sufficient introduction to neural networks to understand this thesis, it is by no means a complete account of the topic. For a more in-depth description of neural networks in general, I refer the readers to Goodfellow et al. (2016). For a primer which is more geared towards NLP, see Goldberg (2015).